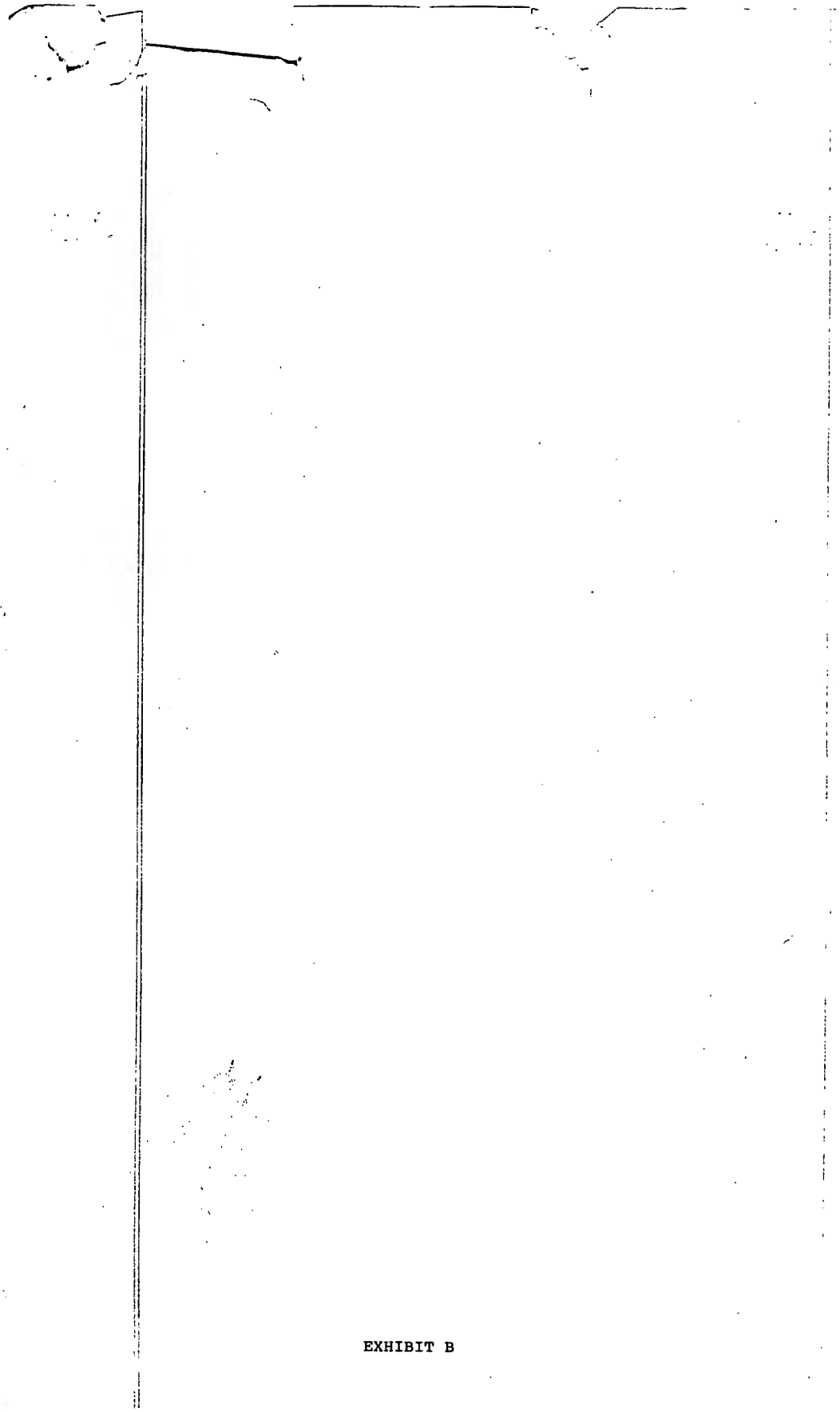
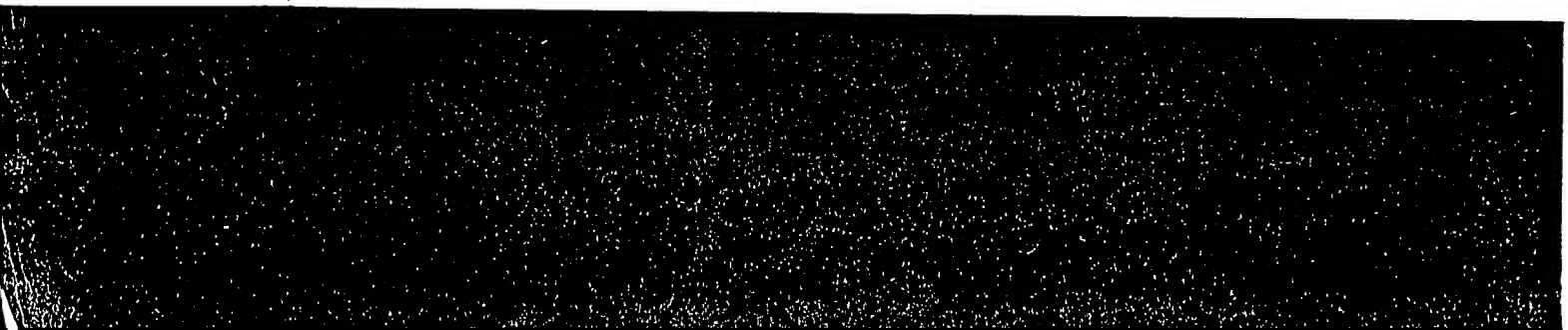


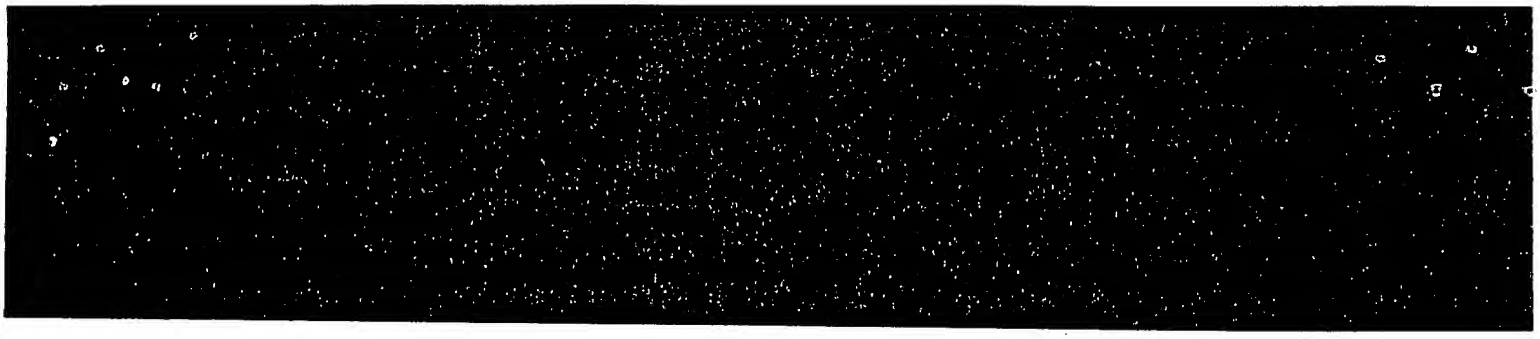
EXHIBIT B

Programmer's Problem Solver for the IBM PC, XT, & AT

Robert Jourdain

Brady Communications Company, Inc.
A Simon&Schuster Publishing Company
New York, New York





Programmer's Problem Solver for the IBM PC, XT, and AT

For information, address Brady Communications Company, Inc.,
Simon&Schuster Building, 1230 Avenue of the Americas, New York, NY 10020.

Library of Congress Cataloging in Publication Data

Jourdain, Robert L., 1950-
Programmer's problem solver for the IBM PC, XT, and AT

Includes index.
1. IBM Personal Computer—Programming. 2. IBM Personal Computer XT—Programming. 3. IBM Personal Computer AT—Programming. I. Title.
QA76.8.I2594J67 1986 005.265 85-14926

ISBN 0-89303-787-7

Printed in the United States of America

86 87 88 89 90 91 92 93 94 95 96 1 2 3 4 5 6 7 8 9 10

3
The Keyboard

Section 1: Monitor the Keyboard

The keyboard contains an Intel microprocessor which senses each keystroke and deposits a *scan code* in Port A of the 8255 peripheral interface chip [1.1.1], located on the system board. A scan code is a one-byte number in which the low seven bits represent an arbitrary identification number assigned to each key. A table of scan codes is found at [3.3.2]. Except in the AT, the top bit of the code tells whether the key has just been depressed (bit = 1, the "make code") or released (bit = 0, the "break code"). For example, the seven-bit scan code of the <B> key is 48, which is 110000 in binary. When the key goes down, the code sent to Port A is 10110000, and when the key is released, the code is 00110000. Thus every keystroke registers twice in the 8255 chip. Each time, the 8255 issues an "acknowledge" signal back to the microprocessor in the keyboard. The AT works slightly differently, sending the same scan code in either case, but preceding it with the byte F0H when the key is released.

When the scan code is deposited in Port A, the keyboard interrupt (INT 9) is invoked. The CPU momentarily sets aside its work and performs a routine that analyzes the scan code. When the code originates from a shift or toggle key, a change in the key's status is recorded in memory. In all other cases the scan code is transformed into a character code, providing it results from a key depression (otherwise the scan code is discarded). Of course, the routine first checks the settings of the shift and toggle keys to get the character code right (is it "a" or "A"?). And then the character code is placed in the *keyboard buffer*, which is a holding area in memory that stores up to fifteen incoming characters while a program is too busy to deal with them. Figure 3-1 shows the path a keystroke takes to travel to your programs.

There are two kinds of character codes, *ASCII codes* and *extended codes*. ASCII codes are one-byte numbers that correspond to the IBM extended ASCII character set, which is listed at [3.3.3]. On the IBM PC, these include the usual typewriter symbols, plus a number of special letters and block-graphics symbols. The ASCII codes also include 32 *control codes* which ordinarily are used to send commands to peripherals, rather than to act as characters on the screen; each, however, has its own symbol which can be displayed by direct memory mapping onto the video display [4.3.1]. (Precisely speaking, only the first 128 characters are true ASCII characters, and it is redundant to speak of "ASCII codes," since "ASCII" stands for "American Standard *Code* for Information Interchange." But programmers commonly speak of "ASCII codes" in order to distinguish them from other numbers.

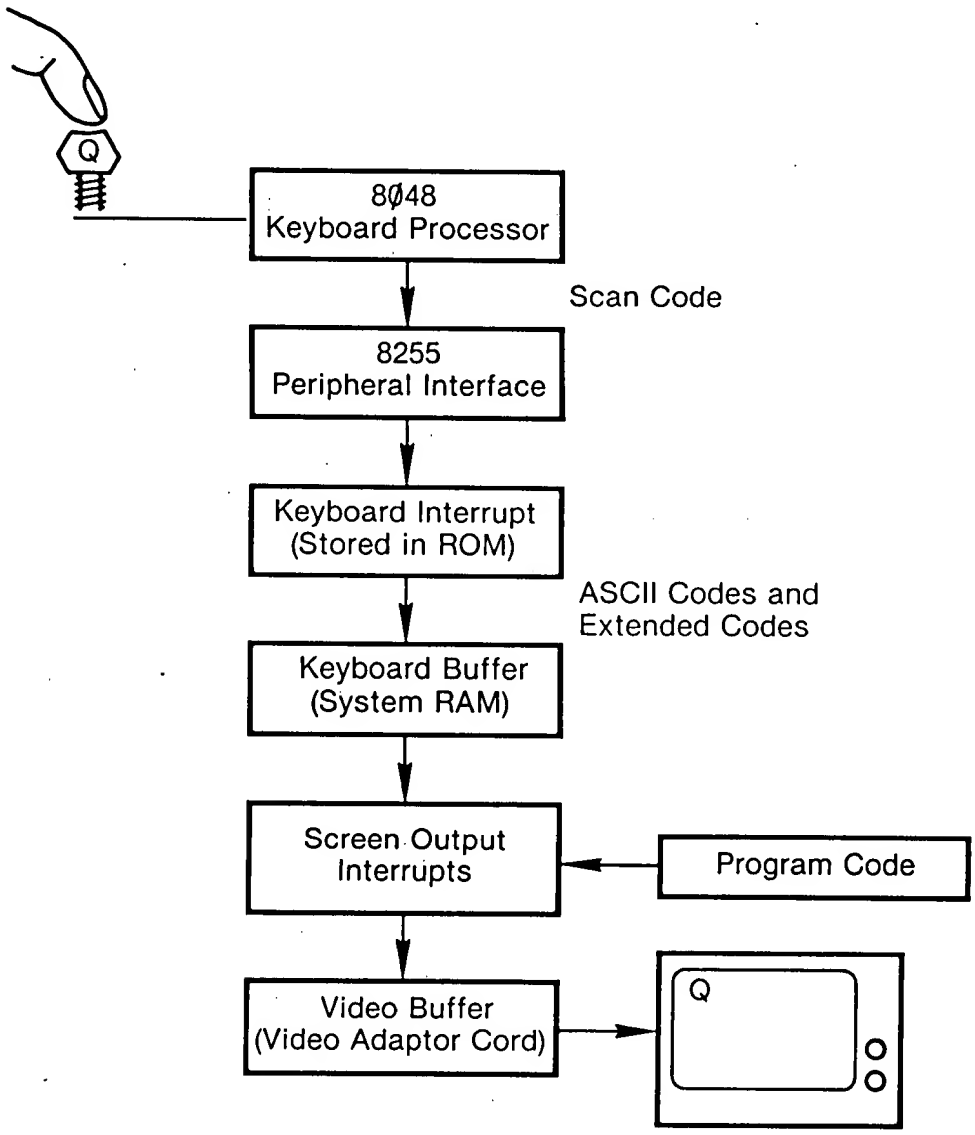


Figure 3-1. From Keyboard To Screen.

For example, "ASCII 8" refers to the backspace character, while "8" is the letter referenced by ASCII 56).

The second kind of codes, the extended codes, are assigned to keys or key-combinations that have no ASCII symbol to represent them, such as the function keys or Alt key combinations. Extended codes are two bytes long, and the first byte is always ASCII Ø. The second byte is a code number, as listed at [3.3.5]. The code Ø;3Ø, for example, represents Alt-A. The initial zero lets programs tell whether a code number is from the ASCII set or the extended set.

There are a few key combinations that perform special functions and that do not generate scan codes. These combinations include <Ctrl-Break>, <Ctrl-Alt-Del>, and <PrtSc>, plus <Sys Req> on the AT, and <Ctrl-Alt-Cursor left, -Cursor right, -CapsLock, -Ins> on the PC Jr. These exceptions bring about special predefined results [3.2.2]. All other keystrokes must be interpreted by your programs, and if they have a special purpose, such as to move the cursor leftward, your program must provide the code that achieves that effect.

Fortunately, the operating system offers a variety of routines that read codes from the keyboard buffer, including means to receive whole strings at once. Because the routines do just about anything you can ask, it is generally senseless to write your own keyboard procedures, and so there are few low level programming examples in this chapter. However, a discussion of how to reprogram the keyboard interrupt is provided.

3.1.1 Clear the keyboard buffer

Programs should clear the keyboard buffer before prompting for input, eliminating any inadvertent keystrokes that may be waiting in the buffer. The buffer holds up to fifteen keystrokes, whether they be one-byte ASCII codes or two-byte extended codes. Thus the buffer must provide two bytes in memory for each keystroke. For one-byte codes, the first byte holds the ASCII code, and the second, the key's scan code. For the extended codes, the first byte holds ASCII 0 and the second byte holds the code number. This code number is usually the key's scan code, but not always, since some keys combine with shift keys to produce more than one code.

The buffer is designed as a *circular queue*, also known as a *first-in first-out (FIFO) buffer*. Like any buffer, it occupies a range of contiguous memory addresses. But no particular memory location is the "front of the line" in the buffer. Rather, two pointers keep track of the 'head' and 'tail' of the string of characters currently in the buffer. New keystrokes are deposited at the position following the tail (towards higher addresses in memory) and the tail pointer is adjusted accordingly. Once the highest memory location of the buffer space is filled, the insertion of new characters wraps around to the low end of the buffer; thus, the head of the string in the buffer will sometimes be at a higher memory location than the tail. Once the buffer is full, additional incoming characters are discarded; the keyboard interrupt beeps the speaker when this happens. Figure 3-2 diagrams some possible configurations of data in the buffer.

While the head pointer points to the first keystroke, the tail pointer points to the position *after* the last keystroke. When the two pointers are equal, the buffer is empty. To allow for fifteen keystrokes, a sixteenth, dummy position is required, and its two bytes always contain a carriage return (ASCII 13), and the scan code for <enter>, which is 28. This dummy position immediately precedes the head of the keystroke string. The 32 bytes of the buffer begin at memory location 0040:00 1E. The head and tail pointers begin at 0040:001A and 0040:001C, respectively. Although the pointers are two bytes long, only the lower, least significant byte is used. The values of the pointers vary from 30 to 60, corresponding to positions within the BIOS data area. Simply set the value of 0040:001A equal to the value in 0040:001C to "clear" the buffer.

Note that it is possible for a program to insert characters into the buffer, ending the string with a carriage return and adjusting the buffer pointers accordingly. If this is done right before exiting a program, when control returns to DOS the characters are read and another program may be loaded automatically.

High Level

In BASIC use PEEK and POKE to fetch and change the values of the buffer pointers:

```
100 DEF SEG=&H40                'set segment to bottom of memory
110 POKE &H1C,PEEK(&H1A)        'equalize the pointers
```


This method
memory, and
in in the mids
leave the buf
discarding th

```
100 IF INK
```

Middle Lev

Function C
7, 8, and A (c
Simply place

```
;---CLEAR
```

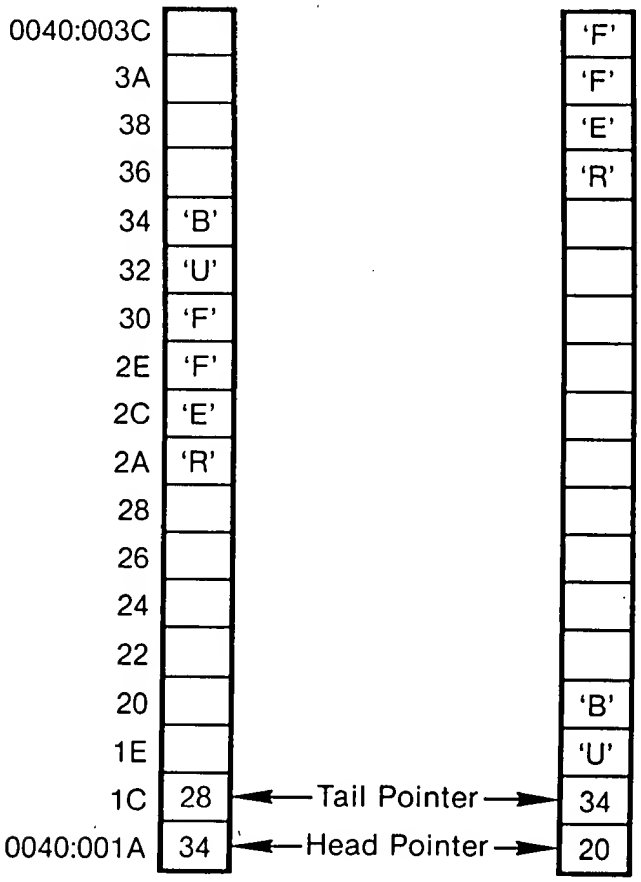


Figure 3-2. Keyboard Buffer Configurations.

This method is not reliable. Some applications may create a buffer elsewhere in memory, and there is also a slight possibility that the keyboard interrupt will break in in the midst of line 110, changing the tail pointer. For these reasons, it is better to leave the buffer pointers alone. Instead, read from the buffer until null is returned, discarding the keystrokes:

```
100 IF INKEY$<>"" THEN 100  'take another keystroke if not null
```

Middle Level

Function C of INT 21H performs any of the DOS keyboard input functions 1, 6, 7, 8, and A (described elsewhere in this section) but clears the keyboard buffer first. Simply place the number of the input function in AL (here it is 1):

```
;---CLEAR BUFFER BEFORE AWAITING KEYSTROKE:
          MOV  AH,0CH          ;select DOS function 0CH
          MOV  AL,1            ;select key input function
          INT  21H             ;clears buffer, waits for keystroke
```

Low Level

As in the high level example, make the tail pointer equal to the head pointer. To avoid interference by the keyboard interrupt, disable interrupts while the change is made:

```
;---EQUALIZE THE HEAD AND TAIL POINTERS:
                CLI                     ;disable interrupts
                SUB   AX,AX             ;make AX=0
                MOV   ES,AX             ;set ES to bottom of memory
                MOV   AL,ES:[41AH]      ;move head pointer to AL
                MOV   ES:[41CH],AL      ;place in tail pointer
                STI                     ;reenable interrupts
```

3.1.2 Check the buffer for keystrokes

You can check whether or not there has been keyboard input without actually removing a character from the keyboard buffer. The buffer uses two pointers that show the front and end of the queue of characters currently in the buffer. When the two pointers are equal, the buffer is empty. Simply compare memory locations 0040:001A and 0040:001C for equality. (One can not merely check for a character at the "front" of the queue, because the buffer is formed as a *circular queue*, and the "front" is constantly changing position [3.1.1]).

High Level

Simply use PEEK to read the two bytes, and compare them:

```
100 DEF SEG=&H40                          'set the memory segment to 0
110 IF PEEK(&H1A)<>PEEK(&H1C) THEN...     '...then a character has arrived...
```

Middle Level

Function B or INT 21H returns FFH in the AL register when the keyboard buffer holds one or more characters, and it returns 00 when the buffer is empty:

```
;---CHECK IF A CHARACTER IS IN THE BUFFER:
                MOV   AH,0BH              ;function number
                INT   21H                 ;call interrupt 21
                CMP   AL,0FFH             ;compare to FF
                JE    GET_KEYSTROKE       ;jump to input routine if char present
```

Function 1 of BIOS interrupt 16H provides the same service, but in addition it shows what the character is. The zero flag (ZF) is set to 0 if the buffer is empty, or to 1 if a character is waiting. In the latter case, a copy of the character at the head of the buffer is placed in AX *without* removing it from the buffer. AL returns the character code for one-byte ASCII characters, or it returns ASCII 0 for extended codes, in which case the code number appears in AH.

```
;---FIND OUT IF THERE IS A CHARACTER:
                MOV   AH,1                ;set function number
                INT   16H                 ;check if character in buffer
                JZ    NO_CHARACTER        ;jump if zero flag = 1
;---THERE IS A CHARACTER, SO SEE WHAT IT IS:
                CMP   AL,0                ;is it an extended code?
                JE    EXTENDED_CODE       ;if so, go to extended code routine
                                          ;otherwise, take character from AL
```

Low Level

As with the high level example, simply compare the two buffer pointers:

```
;---COMPARE HEAD AND TAIL POINTERS:
                MOV   AX,0                ;use the extra segment
                MOV   ES,AX               ;set the segment to 0
                MOV   AL,ES:[41AH]        ;get one pointer
                MOV   AH,ES:[41CH]        ;get other pointer
                CMP   AH,AL               ;compare the pointers
                JNE   GET_KEYSTROKE       ;jump to input routine if unequal
```

3.1.3 Wait for a keystroke and do not echo it on the screen

Normally, incoming keystrokes are echoed on the screen to show what has been typed. But sometimes automatic echoing is undesirable. One-keystroke menu selections need no echo, for example. And sometimes incoming characters may need error-checking before they are sent to the screen. In particular, any program that accepts extended codes must be cautious of automatic echoing, since the first byte (ASCII 0) of these codes will be displayed, leaving spaces between the characters.

High Level

The INKEY$ function of BASIC does not echo. It returns a string that is one byte long for ASCII characters and two bytes long for extended characters. INKEY$ does not wait for a keystroke unless it is placed within a loop that cycles again and again until a character arrives. The loop functions by invoking INKEY$ and then assigning the string it returns to a variable, here C$. When no keystrokes have been received, INKEY$ returns the *null string*, which is a string that is 0 characters long, denoted by two quotation marks with nothing between (""). So long as INKEY$ returns "", the loop repeats: **100 C$ = INKEY$:IF C$ = "" THEN 100.**

The example below assumes that the incoming keystrokes are menu selections and that each selection sends the program to a particular subroutine. The selections are made by striking A,B,C... (resulting in one-byte ASCII codes) or ALT-A, ALT-B, ALT-C... (resulting in two-byte extended codes). To tell the difference, use the LEN function to check whether the string is one or two characters long. If a one-byte ASCII code, a series of IF...THEN statements immediately begin to test the identity of the keystroke, sending the program to the appropriate subroutine. In the case of two-byte codes, control transfers to a separate routine. There the RIGHT$ function eliminates the lefthand character, which, of course, is nothing more than the 0 that identifies extended codes. The ASC function is then used to convert the character from string form to numeric form. Finally, a second series of IF...THEN statements checks the resulting number against those corresponding to ALT-A, ALT-B, etc.

```
100 C$ = INKEY$: IF C$="" THEN 100          'wait for a keystroke
110 IF LEN(C$) = 2 THEN 500                 'if extended code, jump
120 IF C$="a" OR C$="A" THEN GOSUB 1100     'is it menu selection a?
130 IF C$="b" OR C$="B" THEN GOSUB 1200     'b?
140 IF C$="c" OR C$="C" THEN GOSUB 1300     'c?
.
.
500 C$=RIGHT$(C$,1)                         'get 2nd byte of extended code
510 C=ASC(C$)                               'convert to numeric value
520 IF C=30 THEN GOSUB 2100                 'is it menu selection Alt-A?
530 IF C=48 THEN GOSUB 2200                 'Alt-B?
540 IF C=46 THEN GOSUB 2300                 'Alt-C?
```

Note that line 120 (and those following) could instead have used the numeric values for the ASCII codes for "a" and "A", and so on:


```
120 IF C=97 O
```

Of course, first
510. In program
changing C so t
letter. First do s
correct range. T
case. If so, add
shorter statemen

```
500 C=ASC(C$)
510 IF NOT((C>

520 IF C<91 TH
530 IF C=97 TH
```

Middle Level

Functions 7 a
buffer, and wh
Ctrl-Break (and
In both cases, t
extended code h
code appears in

```
;---GET A KEY

;---EXTENDED
 EXTENDED_CO

 C_R:
```

BIOS provid
INT 16H. The
extended codes
extended code n

```
;---GET A KE

;---EXTENDE
 EXTENDED_CC
```

```
120 IF C=97 OR C=65 THEN GOSUB 1100
```

Of course, first C$ would need to be converted to integer form, exactly as in line 510. In programs with a long sequence of these statements, you can save space by changing C so that it always represents either the lower- or upper-case form of a letter. First do some error checking to be sure that the ASCII value of C$ is in the correct range. Then find out if the number is below 91, in which case it is upper case. If so, add 32 to convert it to lower case. Otherwise, do nothing. Then a shorter statement such as IF C = 97 THEN... will suffice. Here is the code:

```
500 C=ASC(C$)                                   'get ASCII number of the character
510 IF NOT((C>64 AND C<91) OR (C>96 AND C<123)) THEN...
                                                '...then out of range, ignore it
520 IF C<91 THEN C=C+32                         'add 32 to value of upper-case letters
530 IF C=97 THEN...                             '... then begin to test the values...
```

Middle Level

Functions 7 and 8 of INT 21H wait for a character if none is in the keyboard buffer, and when one arrives, it is not echoed on the screen. Function 8 detects Ctrl-Break (and initiates the Ctrl-Break routine [3.2.8]), while function 7 does not. In both cases, the character is returned in AL. When AL contains ASCII 0, an extended code has been received. Repeat the interrupt and the second byte of the code appears in AL.

```
;---GET A KEYSTROKE:
                MOV   AH,7              ;set function number
                INT   21H               ;wait for character
                CMP   AL,0              ;see if extended code
                JE    EXTENDED_CODE     ;go to extended code routine if so
                .                       ;otherwise, take character from AL
                .
;---EXTENDED CODE ROUTINE:
 EXTENDED_CODE: INT   21H               ;now the extended code number is in AL
                CMP   AL,75             ;check if "cursor-left"
                JNE   C_R               ;if not, check next possibility
                JMP   CURSOR_LEFT       ;if so, go to routine
 C_R:           CMP   AL,77             ;...etc...
```

BIOS provides a service that matches the DOS function. Place 0 in AH and call INT 16H. The function waits for a character and returns it in AL. In this case, extended codes require calling the interrupt only once. If 0 appears in AL, an extended code number is found in AH. Ctrl-Break is not detected.

```
;---GET A KEYSTROKE:
                MOV   AH,0              ;function number to intercept keystroke
                INT   16H               ;get the keystroke
                CMP   AL,0              ;is it an extended code?
                JE    EXTENDED_CODE     ;if so, go to special routine
                                        ;otherwise, take ASCII char from AL

;---EXTENDED CODE ROUTINE:
 EXTENDED_CODE: CMP   AH,75             ;take extended code from AH
                                        ;...etc...
```

3.1.4 Wait for a keystroke and echo it on the screen

With text or data entry, keystrokes are normally echoed on the screen. In echoing, characters like the carriage return or backspace are interpreted by moving the cursor accordingly rather than displaying the ASCII symbols for the characters. The echoing begins at whatever point the cursor is currently set, and the text automatically wraps around from the last column to the next line. The wrap requires no special coding because the characters are simply deposited at the next position in the video buffer, and the buffer is essentially one long line containing the 25 lines of the screen.

High Level

In BASIC, intercept a keystroke using INKEY$, as shown at [3.1.3]. Then print it before returning to intercept another. Either use the PRINT statement, or else POKE the keystroke directly into the video buffer, using the memory mapping techniques shown at [4.3.1] (the buffer starts at memory segment &HB000 for the monochrome adaptor and at &HB800 for the color adaptor). If you use PRINT, be sure to end the statement with a semicolon, or a carriage return will occur automatically. Below are examples of each method. No attempt is made here to sort out non-character keystrokes. The variable KEYSTROKES$ collects the incoming keystrokes into a data string.

```
100 '''method using PRINT:
110 LOCATE 10,40                        'set the cursor to row 10, col 40
120 KEYSTROKES$=""                      'clear variable that holds incoming string
130 C$=INKEY$:IF C$="" THEN 130         'get a keystroke
140 KEYSTROKES$=KEYSTROKES$+C$          'add the keystroke to a string variable
150 PRINT C$;                           'print the character
160 GOTO 130                            'get next character

100 '''method using POKE (monochrome adaptor):
110 DEF SEG=&HB000                      'set segment offset to start of buffer
120 POINTER=1678                        'position of 10,40 = (2*((10*80)+40))-2
130 KEYSTROKES$=""                      'clear variable holding incoming string
140 C$=INKEY$:IF C$="" THEN 140         'get a keystroke
150 KEYSTROKES$=KEYSTROKES$+C$          'add the keystroke to a string variable
160 POKE POINTER, ASC(C$)               'poke ASCII number of char into buffer
170 POINTER=POINTER+2                   'up pointer by 2 (skip attribute byte)
180 GOTO 140                            'get next character
```

Middle Level

Function 1 of INT 21H waits for a character if none is found in the keyboard buffer, then echos it on the screen at the current cursor position. Ctrl-Break is intercepted so that the (programmable) Ctrl-Break routine is executed [3.2.8]. Characters are returned in AL. In the case of extended codes, AL holds ASCII 0. Repeat the interrupt to bring the second byte of the code into AL.

```
;---GET A KEYSTROKE:
                MOV   AH,1              ;set the function number
                INT   21H               ;wait for a character
```

```
                CMP   AL,0              ;extended code?
                JE    EXTENDED_CODE     ;if so, jump to special routine
                .                       ;else, take ASCII character from AL
                .
;---EXTENDED CODE ROUTINE:
                INT   21H               ;bring the code number into AL
                CMP   AL,77             ;check if "cursor-right"
                JNE   C_R               ;if not, check next possibility
                JMP   CURSOR_RIGHT      ;if so, go to routine
  C_R:          CMP   AL,75             ;...etc...
```

This function completely ignores the escape key. It interprets a tab keystroke normally. The backspace key causes the cursor to move back one space, but the character in that position is not erased. The enter key causes the cursor to move to the start of the current line (there is no automatic line feed).

3.1.5 Intercept a keystroke without waiting

Some real-time applications cannot stop to wait for incoming keystrokes; they take keystrokes from the keyboard buffer only when it is convenient for the program to do so. For example, idling the CPU while awaiting a keystroke would stop all screen action in a video game. Note that it is easy to test whether or not the keyboard buffer is empty, using the methods shown at [3.1.2].

High Level

Simply use INKEY$ without nesting it within a loop:

```
100 C$=INKEY$                   'check for a character
110 IF C$ <> "" THEN...         'there is a character, so...
120 ...                         'else, there is no character
```

Middle Level

Function 6 of INT 21H is the only interrupt that receives keystrokes without waiting. The function does not echo characters on the screen, nor does it sense Ctrl-Break. FFH must be placed in AL before calling this interrupt. Otherwise function 6 serves an entirely different purpose—it prints at the current cursor position whatever character is found in DL. The zero flag is set to 1 if there are no characters in the buffer. When a character is intercepted, it is placed in AL. Should the character be ASCII 0, an extended code is indicated, and a second call is needed to bring in the code number.

```
                MOV   AH,6              ;DOS function 6
                MOV   DL,0FFH           ;request function for keyboard input
                INT   21H               ;get character
                JZ    NO_CHAR           ;jump to NO_CHAR if no keystroke
                CMP   AL,0              ;see if character is ASCII 0
                JE    EXTENDED_CODE     ;if so, go to extended code routine
                ...                     ;ASCII character now in AL

EXTENDED_CODE:  INT   21H               ;get 2nd byte of extended code
                ...                     ;code number now in AL
```

3.1.9 Reprogram the keyboard interrupt

When the keyboard microprocessor deposits a scan code in Port A of the 8255 chip (at port address 60H—see [1.1.1]), it invokes INT 9. The job of this interrupt is to convert the scan code to a character code on the basis of the shift and toggle key settings, and to place the code in the keyboard buffer. (When the scan code is for a shift or toggle key, no character code goes to the buffer (except for <Ins>); instead, the interrupt makes changes in two status bytes located in the BIOS data area [3.1.7]). The BIOS and DOS "keyboard interrupts" are really only "keyboard buffer interrupts." They do not actually "read" keystrokes. Rather, they read the *interpretations* of keystrokes that INT 9 provides. Note that the PCjr uses a special routine (INT 48H) to convert input from its 62 keys into the 83-key protocol used by the other IBM machines. The results of this routine are passed on to INT 9, which performs its work as usual. Via INT 49H, the PCjr also provides for special non-key scan codes that could potentially be set up for peripheral devices that would make use of the infrared (cordless) keyboard link.

It takes a very unusual application to make it worthwhile to reprogram this interrupt, especially considering that DOS allows you to reprogram any key of the keyboard [3.2.6]. Still, if you must reprogram INT 9, this section will give you a start. Read [1.2.3] first to understand in general how interrupts are programmed. There are three basic steps in the keyboard interrupt:

1. Read a scan code and send an *acknowledge* signal to the keyboard.
2. Convert the scan code into a code number or into a setting in the shift/toggle key status register.
3. Place a key code in the keyboard buffer.

At the time the interrupt is invoked, a scan code will be in Port A. So first the code is read and saved on the stack. Then Port B (port 61H) is used to very briefly issue the "acknowledge" signal to the keyboard microprocessor. Simply change bit 7 to 1, then immediately change it back to 0. Note that bit 6 of Port B controls the clock signal of the keyboard. It must always be 1, or the keyboard is effectively turned off. These port addresses apply to the AT as well, even though it does not have an 8255 interface chip.

The scan code is first analyzed to see whether the key was depressed (the "make" code) or released (the "break" code). Except on the AT, a break code is indicated when bit 7 of the scan code is set to 1. On the AT, where bit 7 is always 0, a break code is two bytes: first 0F0H and then the scan code. All break codes are thrown away except those for shift and toggle keys, for which the appropriate changes are made in the shift/toggle status bytes. On the other hand, all make codes are processed. Here again the shift/toggle status may be changed. But in the case of character codes, the status bytes must be consulted to see whether, for example, the scan code 30 indicates an upper or lower case A.

Once an incoming character has been identified, the keyboard routine must find its ASCII code or extended code. The example here is much too short to show all cases. In general, the scan code is correlated with an entry in a data table that is accessed by the XLAT instruction. XLAT takes a number from 0-255 in AL and

returns in AL a corresponding one-byte value from a 256-byte table that is pointed to by DS:BX. The table may be set up in the data segment. If AL initially contains scan code 30, then AL receives byte number 30 of the table (the *31st* byte, since we're counting from 0). This byte of the table should have been set to 97, giving the ASCII code for a. Of course, a second table would be required for capital A, and it would be called instead should the routine find that the shift state is "on". Or, alternatively, some other part of a single table could hold the capital letters, in which case the scan code would need to have an offset added to it.

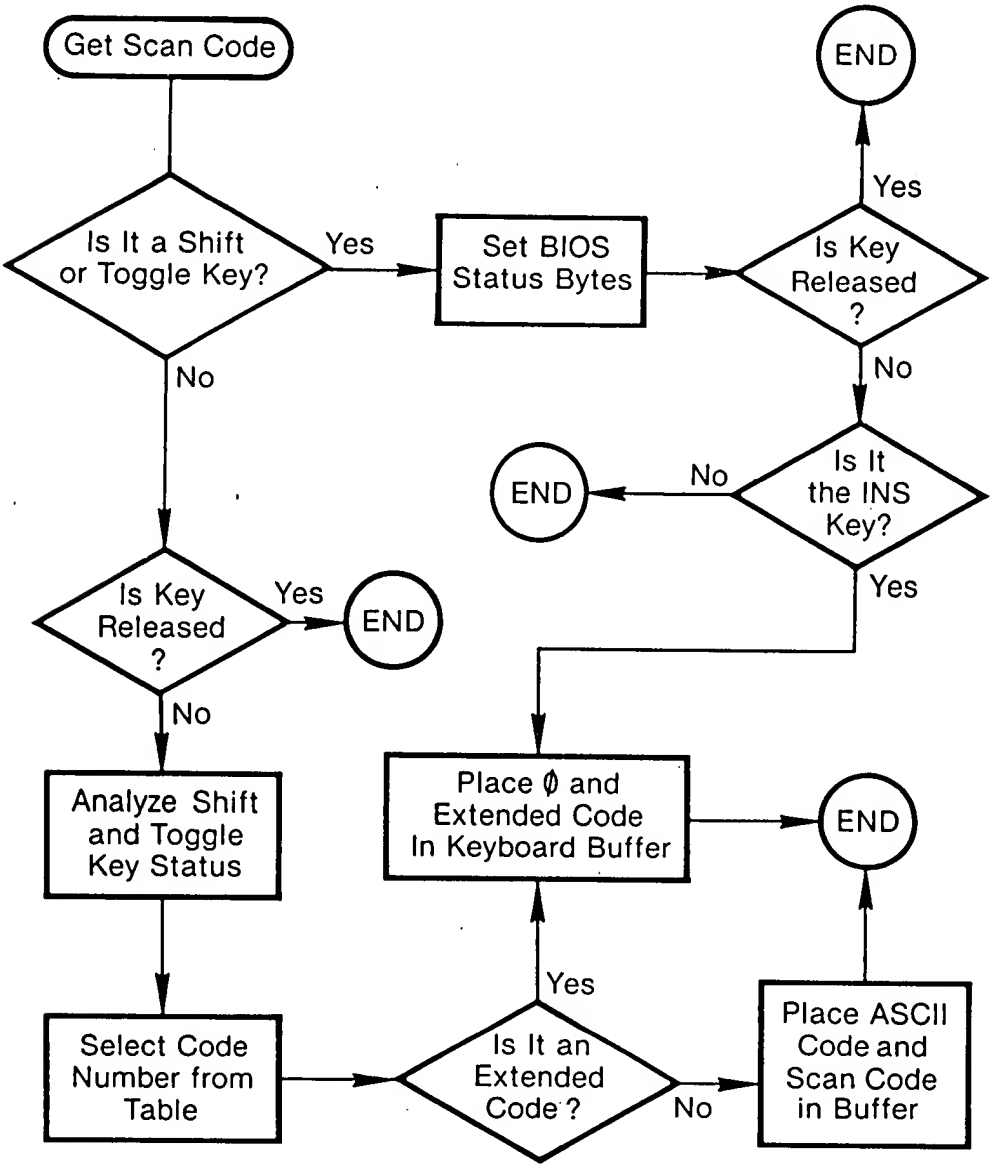
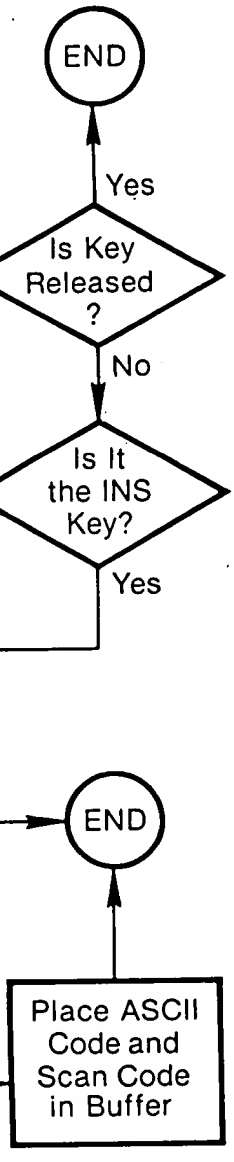


Figure 3-4. The Keyboard Interrupt.

able that is pointed
L initially contains
he *31st* byte, since
set to 97, giving the
or capital A, and it
e is "on". Or, alter-
al letters, in which



Finally, code numbers must be placed in the keyboard buffer. The routine must first check to see if there is any room in the buffer for another character. [3.1.1] shows how the buffer is constructed as a *circular queue*. Memory location 0040:001A contains the pointer for the head of the buffer, and 0040:001C contains the pointer for the tail. The word-length pointers are offsets within the BIOS data area (which starts at segment 40H), ranging from 30 to 60. New characters are inserted at higher memory positions in the buffer, and when the upper limit is reached, the insertion wraps around to the low end of the buffer. When the buffer is full, the tail pointer is 2 less than the head pointer—except when the head pointer equals 30 (is at the top of the buffer) in which case the buffer is full if the value of the tail pointer is 60.

To insert a character in the buffer, place it at the position pointed to by the tail pointer, then increase the tail pointer by 2; if the tail pointer equals 60, change it instead to 30. That is all there is to it. Figure 3-4 diagrams the keyboard interrupt.

Low Level

An efficient routine requires much thought. This example gives only the rudiments. It intercepts only the lower- and upper-case letters, loading them both into the same table, with the capital letters 100 bytes higher in the table than their siblings. Only the left shift key is attended to, and the current status of the CapsLock is ignored.

```
;---IN THE DATA SEGMENT:
   TABLE          DB 16 DUP(0)                ;skip first 16 bytes of table
                  DB 'quertyuiop',0,0,0,0     ;top row (scan code #16 = q)
                  DB 'asdfghjkl',0,0,0,0,0;middle row
                  DB 'zxcvbnm'                ;bottom row
                  DB 16 dup(0)                ;offset upper case to 100 bytes higher
                  DB 'QUERTYUIOP',0,0,0,0     ;caps for top row
                  DB 'ASDFGHJKL',0,0,0,0,0;caps for middle row
                  DB 'ZXCVBNM'                ;caps for bottom row

;---AT BEGINNING OF THE PROGRAM, INSTALL THE INTERRUPT:
                  CLI                         ;disable interrupts
                  PUSH DS                     ;save DS
                  MOV  AX,SEG NEW_KEYBOARD ;make DS:DX point to interrupt routine
                  MOV  DS,AX
                  MOV  DX,OFFSET NEW_KEYBOARD
                  MOV  AL,9                   ;number of interrupt vector to change
                  MOV  AH,25H                 ;DOS function to change vector
                  INT  21H                    ;change the vector
                  POP  DS                     ;restore DS
                  STI                         ;reenable interrupts
                  .
                  .
```

(The program continues, perhaps ending and staying resident [1.3.4])

```
;---HERE IS THE KEYBOARD INTERRUPT ITSELF:
NEW_KEYBOARD      PROC FAR                    ;hardware interrupts are far procedures
                  PUSH AX                     ;save all changed registers
                  PUSH BX                     ;
                  PUSH CX                     ;
```